

Interactive Bike Computer

Interactive-bike.com

Shane Heavey

BEng (H) Software and Electronic Engineering

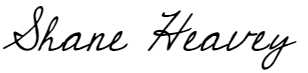
Galway-Mayo Institute of Technology

2019/2020

**Declaration**

It is my great pleasure to declare that the project entitled ***Interactive Bike Computer*** submitted to Galway-Mayo Institute of Technology in partial fulfilment of the requirements for the degree of BEng (H) Software & Electronic Engineering is my original work.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.





**Acknowledgments**

I would like to acknowledge my supervisor Des O’Reilly for his continuous help and guidance with this project.

He helped me with challenges and questions I had throughout the course of this project.

## Table of Contents

[1.1 Table of Contents 4](#_Toc40615377)

[2 Summary 7](#_Toc40615378)

[3 Interactive Bike Computer Poster 8](#_Toc40615379)

[4 Introduction 9](#_Toc40615380)

[4.1 Project Goals 9](#_Toc40615381)

[4.2 Project Motivation 9](#_Toc40615382)

[4.3 Report Overview 9](#_Toc40615383)

[5 History of Bike Computers 10](#_Toc40615384)

[6 Project Architecture 11](#_Toc40615385)

[7 Images of project 12](#_Toc40615386)

[8 Hardware 13](#_Toc40615387)

[8.1 Raspberry Pi 13](#_Toc40615388)

[8.2 Crontab command 13](#_Toc40615389)

[8.3 Hall Effect Sensor 14](#_Toc40615390)

[8.3.1 Hall effect sensor Wiring Diagram 14](#_Toc40615391)

[9 MQTT 15](#_Toc40615392)

[9.1 How I am sending the MQTT data 15](#_Toc40615393)

[9.1.1 To the website 15](#_Toc40615394)

[9.1.2 To the Android Application 15](#_Toc40615395)

[10 Python script 16](#_Toc40615396)

[10.1 Calculations 16](#_Toc40615397)

[10.1.1 Current Speed 16](#_Toc40615398)

[10.1.2 Average Speed 16](#_Toc40615399)

[10.1.3 Maximum Speed 16](#_Toc40615400)

[10.1.4 Distance 17](#_Toc40615401)

[10.1.5 Revolution per minute (RPM) 17](#_Toc40615402)

[11 Android Studio 18](#_Toc40615403)

[11.1 Why Android Studio? 18](#_Toc40615404)

[11.2 Flowchart of Android application 18](#_Toc40615405)

[11.3 Main Activity 19](#_Toc40615406)

[11.3.1 Toast Overview 19](#_Toc40615407)

[11.4 Second activity 20](#_Toc40615408)

[11.4.1 Code snippet 20](#_Toc40615409)

[11.5 Map Activity 21](#_Toc40615410)

[11.6 Speed Activity 22](#_Toc40615411)

[11.6.1 Connecting the MQTT client to the broker 22](#_Toc40615412)

[11.6.2 Subscribing to a topic 23](#_Toc40615413)

[11.6.3 Receiving a topic 24](#_Toc40615414)

[11.6.4 Analog Speedometer 25](#_Toc40615415)

[11.7 Graph Activity 25](#_Toc40615416)

[11.7.1 DBHelper class 26](#_Toc40615417)

[12 Website 27](#_Toc40615418)

[12.1 Goal 27](#_Toc40615419)

[12.2 Navigating the website 27](#_Toc40615420)

[12.3 Images of the website 27](#_Toc40615421)

[12.4 Amazon Web Services 31](#_Toc40615422)

[12.4.1 Connecting my website to AWS server 31](#_Toc40615423)

[12.4.2 Domain name (DNS) 32](#_Toc40615424)

[12.4.3 Connecting NodeJS & MongoDB to AWS server 32](#_Toc40615425)

[12.5 Full stack development 33](#_Toc40615426)

[12.5.1 MVC Framework 34](#_Toc40615427)

[12.5.1.1 Advantages of MVC 34](#_Toc40615428)

[12.6 Backend 35](#_Toc40615429)

[12.6.1 Database & Models 35](#_Toc40615430)

[12.6.2 Routes 37](#_Toc40615431)

[12.6.3 Controllers 38](#_Toc40615432)

[12.6.4 View 38](#_Toc40615433)

[12.6.5 MQTT website connection 39](#_Toc40615434)

[12.6.6 Backend diagram 40](#_Toc40615435)

[12.7 Frontend 41](#_Toc40615436)

[12.7.1 jQuery & Ajax 41](#_Toc40615437)

[12.7.2 Chart.js 42](#_Toc40615438)

[12.7.3 Bootstrap 43](#_Toc40615439)

[12.7.4 Select2.js 45](#_Toc40615440)

[13 System Integration 46](#_Toc40615441)

[14 Conclusion 47](#_Toc40615442)

[15 References 48](#_Toc40615443)

[16 Appendices 50](#_Toc40615444)

[16.1 Bill of Materials 50](#_Toc40615445)

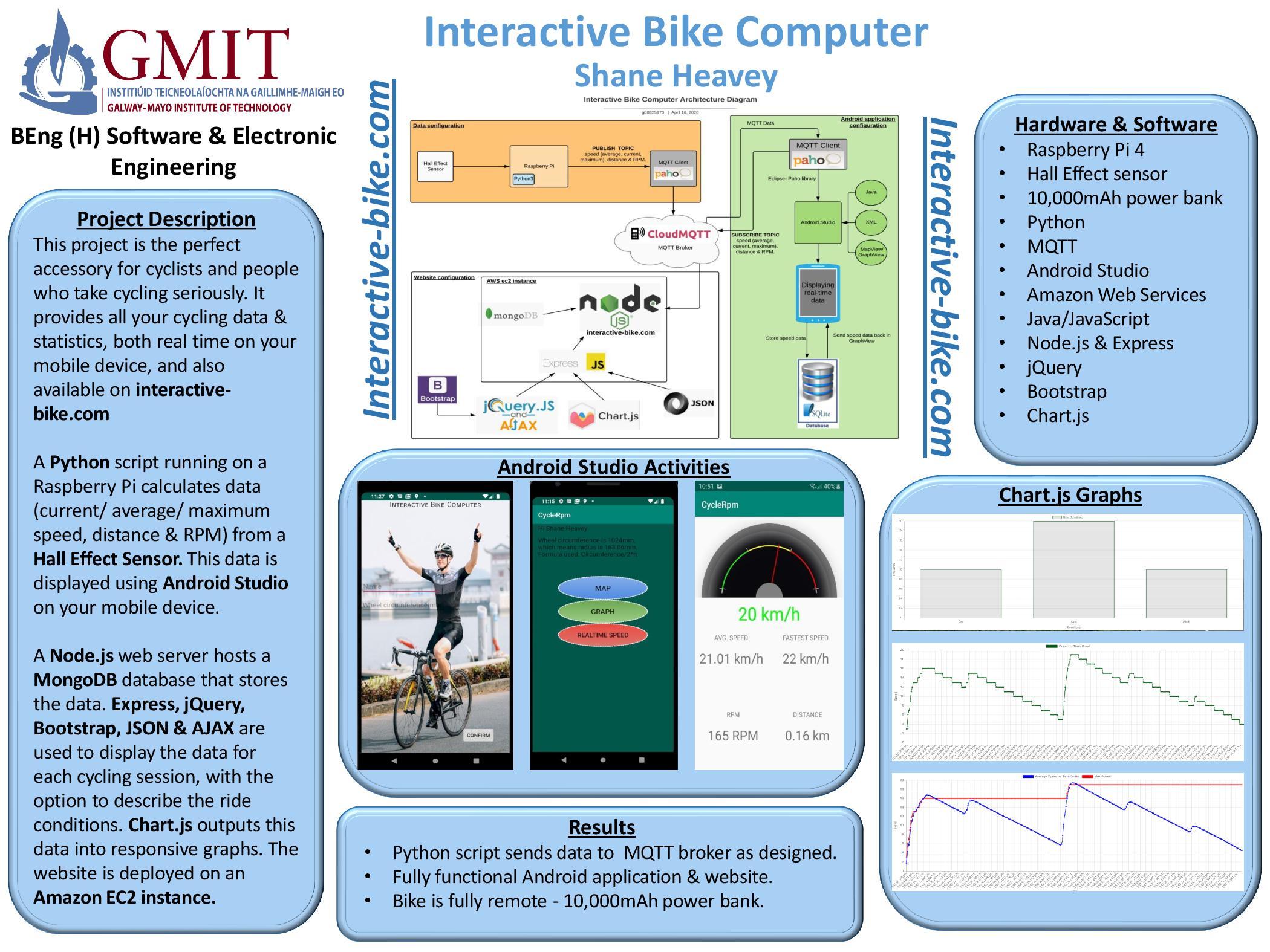
# Summary

The outcome of this project is a multi-functional bike computer that calculates data to which a cyclist would want to have real-time access. This project is geared towards professional cyclists and people who take cycling seriously. During the research phase, numerous avid cyclists were asked their opinions, with the consensus being that everyone would enjoy having access to real-time statistics while cycling.

The scope of the project begins with collecting data, such as current, average, and maximum speed, distance, and revolutions per minute (RPM). It then transmits data from a sensor to an MQTT broker for storage, where a user-friendly website and Android application that subscribe to the data display it in real-time.

The project reached its goal of developing a fully functional module for cyclists. The project was challenging and presented numerous learning opportunities. Some of the learning opportunities include the use of new hardware and programming languages, including, but not limited to, Python, Java, and JavaScript, Node, Express, jQuery, chart.js, and ajax.

# Interactive Bike Computer Poster



# Introduction

## Project Goals

* To research, design, build, and program a fully functioning website and Android application.
* To combine the Hall effect sensor and Python algorithms to get an output for current/average/maximum speed, distance, and RPM.
* To learn more about app development, AWS deployment, graph.js and other methodologies such as AJAX, JSON, and Express.js.
* To have all components of the project fully operational.

## Project Motivation

With a passion for sports, and more specifically, cycling, the idea for this project is of great personal interest. Having real-time statistics while cycling is very beneficial. The target audience for this project are professional cyclists and people who take cycling seriously.

There is a need for real-time statistics while cycling.

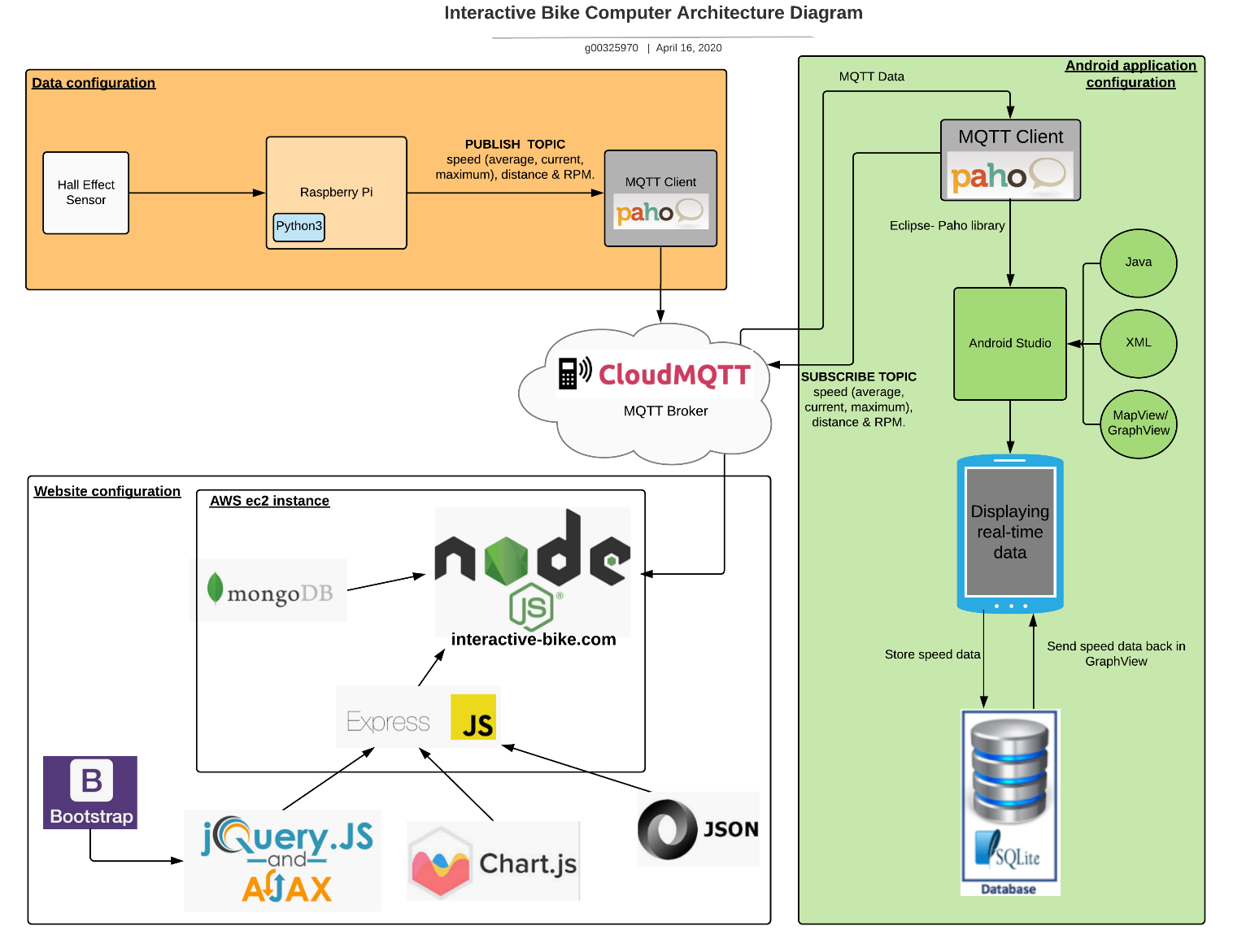
## Report Overview

My report is written in sections including the project idea, project architecture. The body of the report details the principal areas of the project. I have also added images of components used and schematics. Snippets of test code and library names are furthermore included throughout the report.

# History of Bike Computers

In 1895, Curtis H. Veeder invented the Cyclometer. This simple mechanical device counted the number of rotations of a bicycle wheel. Veeder’s slogan was ‘*It's Nice to Know How Far You Go’* **[1].** It is great to see this slogan is relevant more than ever in the world we live in today. People are working with technologies every day and also want to incorporate them into their pastimes. From my research, what Mr. Veeder seen in developing this mechanical device by back, is still very much relevant today. I am sure he would be happy to see how far the technology has progressed in this area.

# Project Architecture



**Figure 6-1 Architecture Diagram**

# Images of project



Raspberry Pi 4

Hall effect sensor

Battery pack



Recommended area to display phone

# Hardware

## Raspberry Pi

The Raspberry Pi is a low cost, credit-card sized computer that plugs into a computer monitor or TV. It uses a standard keyboard and mouse with one output pin along with 5v/GND to power the Hall effect sensor. This sensor is attached to the fork of the wheel. A magnet is attached to the spoke of the wheel and can enter the magnetic field of the sensor every time it rotates **[2].**



**Figure 8-1** Raspberry Pi 4

## Crontab command

A crontab command runs a Python script on start-up of the Raspberry Pi 4. Cron is a tool for configuring scheduled commands or scripts to run at periodic or fixed intervals on Unix systems. The command crontab (cron table) is used to edit the list of scheduled tasks in operation on a per-user basis; each user (including root) has a crontab. The command ‘sudo crontab -e’ opens up the cron table. After testing and research, a delay within the startup file worked best as both the Raspberry Pi and Python script were trying to load at the same time. This delay lets the Pi boot up first and connect to the hotspot on the phone. As it is not feasible for the cyclist to have a keyboard and monitor with them on their ride, this command is ideal for the project **[3].**

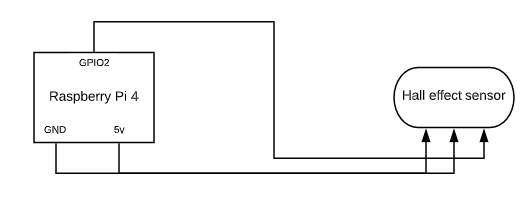
## Hall Effect Sensor

Hall effect sensors are devices that are activated by an external magnetic field. Within the sensor, a thin strip of metal has a current applied along with it. Charge carriers will then travel along in a straight line. In the presence of a magnetic field, the charge carriers will deviate from their straight-line path and deflect towards an edge, producing a voltage gradient. This is called the Lorentz force and is what triggers the sensor **[4].**



**Figure 8-3** Hall effect sensor

### Hall effect sensor Wiring Diagram



**Figure 8-3-1** Sensor Circuit Diagram

# MQTT

MQTT stands for Messaging Queue Telemetry Transport. It is a TCP/IP based, publish and subscribe messaging protocol, designed for lightweight machine to machine learning. All clients open a TCP connection to the broker. MQTT does not use addresses but uses a subject line called ‘Topics.’ Cloudmqtt.com is used as the broker. [5]



## How I am sending the MQTT data

### To the website

The data is converted into JSON before it’s sent across as JSON is more compatible and easier to use in terms of the website’s frontend.

all\_data = json.dumps({"d\_id" : device\_id, "speed" : str(int(km\_per\_hour)), "rpm" : str(int(rpm)), "distance" : str(distance), "average" : str(avgSpeed\_n), "maximum" : str(int(MaxSpeed))})

mqttc.publish('all\_data', all\_data)

### To the Android Application

The MQTT publish command is used to send all the messages to the broker. The data is typecast, depending on whether it needs to be an integer or a float.

mqttc.publish("Speed",int(km\_per\_hour))

mqttc.publish("RPM",int(rpm))

mqttc.publish("MSpeed",int(MaxSpeed))

mqttc.publish("ASpeed","%.2f"%avgSpeed)

mqttc.publish("Distance","%.2f"%float(int(dist\_meas)/1000))

# Python script

The purpose of this Python script is to manipulate the data received from the hall effect sensor. Python was the chosen language as it was already used for a module called Digital Signal Processing (DSP).

## Calculations

The desired data output was current/ average/ maximum speed, the distance traveled, and RPM.

### Current Speed

After finding the circumference of the wheel in centimeters, it wasdivided by 10,000 to convert from centimeters to kilometers. The value was then divided by the elapsed time to get kilometers per second. The elapsed time is the time taken for one complete rotation of the wheel. Finally, the output was multiplied by 3600 to convert from kilometers per second (km/s) to kilometers per hour (km/h).

circ\_cm = (2\*math.pi)\*r\_cm # calculate wheel circumference in CM

dist\_km = circ\_cm/100000 # convert cm to km

km\_per\_sec = dist\_km / elapse # calculate km/sec

km\_per\_hour = km\_per\_sec \* 3600 # calculate km/h

### Average Speed

A new variable called avgSpeed was created and initialized to 0. The new variable had kilometers per hour added then was divided by 2. This same calculation would happen every time a new MQTT message was delivered.

avgSpeed=(avgSpeed+km\_per\_hour)/2

### Maximum Speed

A new variable called MaxSpeed was created and initialized to 0. An if statement was used to check if kilometers per hour was greater than MaxSpeed.

if(km\_per\_hour>MaxSpeed):

MaxSpeed=km\_per\_hour

### Distance

Distance is simply speed \* time. The value is then divided by 1000. The output can have many values after 0, so it is then restricted to two decimal places.

dist\_meas = (dist\_km\*pulse)\*1000 # measure distance traverse in meter

### Revolution per minute (RPM)

RPM was calculated by dividing one by the elapsed time. The output was multiplied by 60 to get the revolutions in minutes.

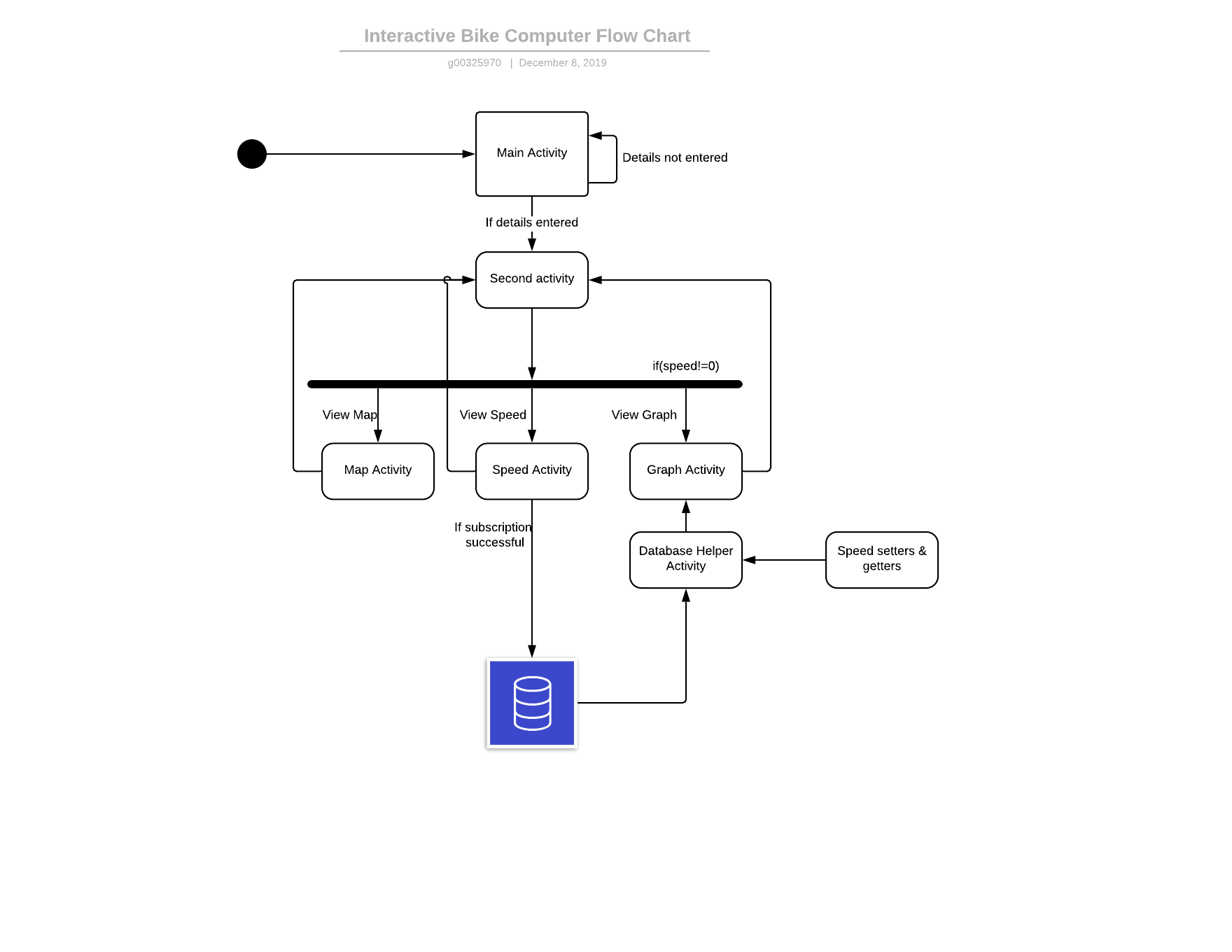
rpm = 1/elapse \* 60

# Android Studio

## Why Android Studio?

Android Studio is the official integrated development environment for Google's Android operating system. It’s designed specifically for Android development and is built on JetBrains' IntelliJ IDEA software. Along with Android Studio providing the fastest tools for building apps on every type of Android device, it was also a module on the syllabus the same year as this final year project **[6]**.

## Flowchart of Android application



**Figure 11-1** Flow Chart for Android Application

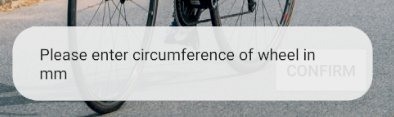
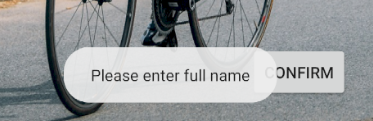
## Main Activity

The main activity is the first page the user will see when opening the app. It greets the user with a relevant picture and asks for a name and the wheel circumference of the wheel in millimetres. This is a requirement for calculating the speed. Once required parameters are interest, the next accessed. This happens using intents. (See 6)

### Toast Overview

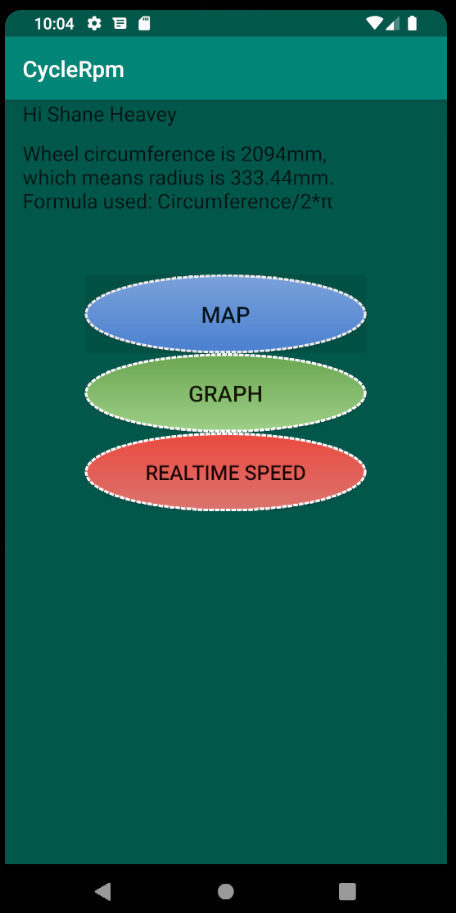
The application cannot continue until both details are input. A toast overview is sent if the user tries to progress without this information.

button.setOnClickListener(**new** View.OnClickListener() {  
 @Override  
 **public void** onClick(View v) {  
 **if** (isEmpty(editText1)) {  
 Toast.*makeText*(getApplicationContext(), **"Please enter full name"**, Toast.***LENGTH\_SHORT***).show();  
 **return**;  
 }  
 **if**(isEmpty(editText2)){  
 Toast.*makeText*(getApplicationContext(), **"Please enter circumference of wheel in mm"**, Toast.***LENGTH\_SHORT***).show();  
 **return**;  
 }  
  
 openActivity2();  
 }  
});



**Figure 11-3-1** Toast authentication

## Second activity



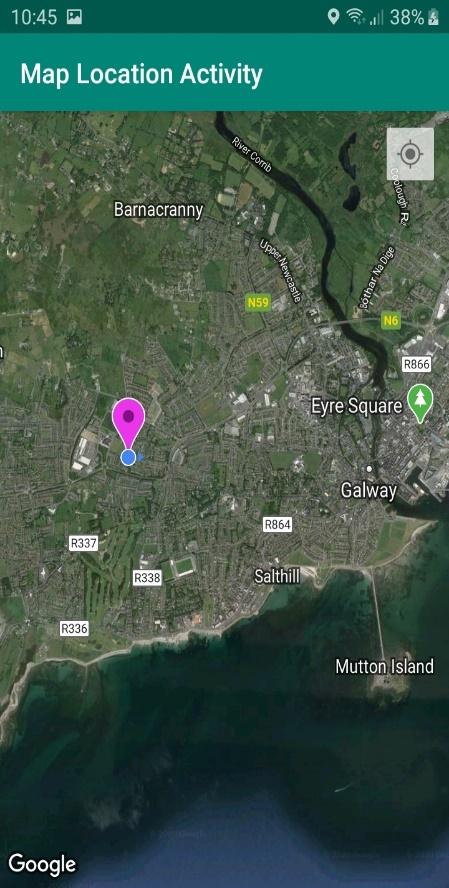
The second activity acts as a gateway to the rest of the app. It starts by greeting the user and displays the entered wheel circumference of the bike along the conversion to the radius. The formula is shown underneath along with. I customized buttons, which make the app more user-friendly. Intents is used to open the desired activity.

### Code snippet

@Override  
**public void** onClick(View v) {  
 **switch** (v.getId()) {  
 **case** R.id.***graph***:  
 openGraphActivity();  
 **break**;  
 **case** R.id.***map***:  
 openMapActivity();  
 **break**;  
 **case** R.id.***speed***:  
 openSpeedActivity();  
 **break**;  
 }  
}

A simple switch statement waits for a button to be pressed, then opens a function specific to that button. All three functions use intents to open their specific activity.

## Map Activity



The map activity consists of a simple to use map that pinpoints the exact area the rider is at that moment. The user can zoom in and out as needed. This option was included because a cyclist might be riding in an area they are not familiar with and would like the option to view on a map, be it looking for a smoother road or the scenic route.

This map feature was added by installing MapView from the Google Play services SDK and getting a Google Maps API Key. An API key is a unique identifier that you generate using the Google Cloud Platform Console. A View which displays a map (with data obtained from the Google Maps service). When focused, it will capture keypresses and touch gestures to move the map. **[7] [19].**

## Speed Activity



The data is shown in real-time within the speed activity in a user-friendly and easy-to-use manner. The cyclist will be looking at this activity the most while on their ride, so everything should be clear and precise. The analog speedometer moves in line with the number displayed just below it. Average speed, fastest speed, RPMs, and distance also adjust accordingly.

### Connecting the MQTT client to the broker

The credentials of the broker are added for the client in this activity to connect. Once the activity is opened, it tries to connect to the cloudmqtt.com broker.

**client** =  
 **new** MqttAndroidClient(**this**.getApplicationContext(), **"tcp://farmer.cloudmqtt.com:11836"**,  
 clientId);  
MqttConnectOptions options = **new** MqttConnectOptions();  
options.setMqttVersion(MqttConnectOptions.***MQTT\_VERSION\_3\_1***);  
options.setUserName(**"username"**);  
options.setPassword(**"password"**.toCharArray());

### Subscribing to a topic

The subscribe method accepts two parameters – A topic or topics and a QOS (Quality of Service). This method guarantees that the message will be transferred successfully to the broker. Setting QOS to 1 means the client will publish data to the broker and receive an acknowledgement back from the broker. QOS is set to 1 on the Raspberry Pi by default. The client then begins to subscribe to the MQTT topic ‘speed’. Toast overviews have been implemented that let the user know if a subscription has occurred successfully or not. Code that might throw an exception is enclosed in a try-catch.

**final** String topic = **"Speed"**;  
**int** qos = 1; *//quality of service***try** {  
 IMqttToken subToken = **client**.subscribe(topic, qos);  
 **client**.subscribe(**"MSpeed"**,1);  
 **client**.subscribe(**"RPM"**,1);  
 **client**.subscribe(**"Distance"**,1);  
 **client**.subscribe(**"ASpeed"**,1);  
  
 subToken.setActionCallback(**new** IMqttActionListener() {  
 @Override  
 **public void** onSuccess(IMqttToken asyncActionToken) {  
 *// successfully subscribed* Toast.*makeText*(ActivitySpeed.**this**, **"Successfully subscribed to: "** + topic, Toast.***LENGTH\_SHORT***).show();  
  
 }  
  
 @Override  
 **public void** onFailure(IMqttToken asyncActionToken,  
 Throwable exception) {  
 *// The subscription could not be performed, maybe the user was not  
 // authorized to subscribe on the specified topic e.g. using wildcards* Toast.*makeText*(ActivitySpeed.**this**, **"Couldn't subscribe to: "** + topic, Toast.***LENGTH\_SHORT***).show();  
  
 }  
 });  
} **catch** (MqttException e) {  
 e.printStackTrace();  
} **catch** (NullPointerException e) {  
 e.printStackTrace();  
}

### Receiving a topic

Once the topic is received, it is assigned to its personal XML TextView. This is how it knows where it is to be displayed on the application.

**if** (topic.equals(**"Speed"**)) {  
 TextView val2 = (TextView) findViewById(R.id.***value***);  
 Log.*d*(**"speed"**, message.toString());  
 val2.setText(message.toString() + **" km/h"**);  
 **speedometer**.setSpeed(Double.*parseDouble*(message.toString()),**true**);  
 Long tsLong = System.*currentTimeMillis*()/1000;  
 String ts = tsLong.toString();  
 DBHelper db = **new** DBHelper(**this**);  
 *// Inserting Contacts* Log.*d*(**"Insert: "**, **"Inserting .."**);  
 db.addContact(**new** Speed(message.toString(), ts));  
}  
 **if** (topic.equals(**"ASpeed"**)) {  
 TextView val = (TextView) findViewById(R.id.***Aspeed***);  
 Log.*d*(**"Aspeed"**, message.toString());  
 val.setText(message.toString() + **" km/h"**);  
 **int** ival = Integer.*parseInt*(message.toString());  
 }  
 **if** (topic.equals(**"MSpeed"**)) {  
 TextView val = (TextView) findViewById(R.id.***Mspeed***);  
 Log.*d*(**"Mspeed"**, message.toString());  
 val.setText(message.toString() + **" km/h"**);  
 **int** ival = Integer.*parseInt*(message.toString());  
  
  
 }  
 **if** (topic.equals(**"Distance"**)) {  
 TextView val = (TextView) findViewById(R.id.***distance***);  
 Log.*d*(**"Distance"**, message.toString());  
 val.setText(message.toString() + **" km"**);  
 **int** ival = Integer.*parseInt*(message.toString());  
 }  
 **if** (topic.equals(**"RPM"**)) {  
 TextView val = (TextView) findViewById(R.id.***RPM***);  
 Log.*d*(**"RPM"**, message.toString());  
 val.setText(message.toString() + **" RPM"**);  
 **int** ival = Integer.*parseInt*(message.toString());  
 }

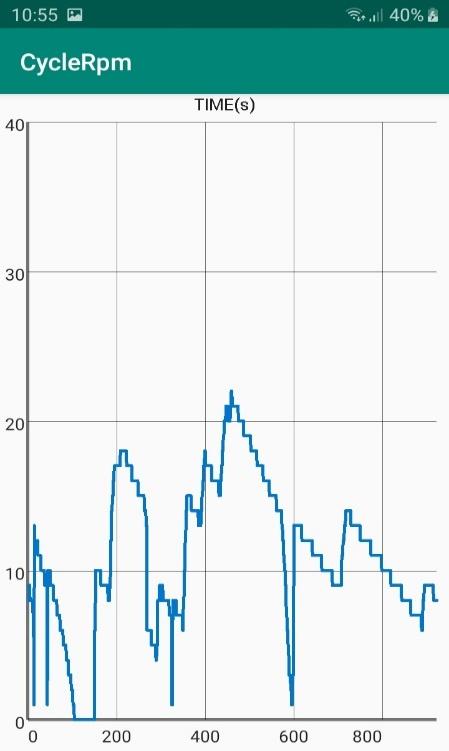
Sending speed value to DBHelper. For displaying in the graph activity. Converted to seconds.

### Analog Speedometer



The cyclist may only be able to look at their speed for a split-second while on the road. The main goal with this activity was to make it very easy to process the data. This simple speedometer gauge was found during research. [8]

## Graph Activity



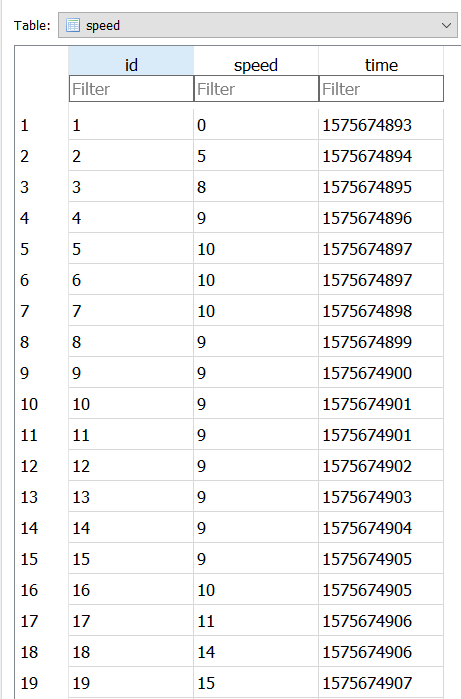
**Note:** Graph is somewhat sporadic as it was tested indoors

The data shown in the graph activity is a Speed vs. Timeline chart. Although you can see this data on the website, it is ideal for the user to get a quick visual representation of their speed throughout their bike ride. They might not have time to check the website until a later date and can easily enter this activity to see the overall speed performance of their bike ride.

### DBHelper class

To plot this graph, the data was stored in a local database using SQLite. This DBHelper class was created and SQLiteOpenHelper extended, which is a helper class to manage database creation and version management. For performing any database operation, you have to provide the implementation of onCreate() and onUpgrade() methods of SQLiteOpenHelper class **[9]**.

*// Creating Tables*@Override  
**public void** onCreate(SQLiteDatabase db) {  
 String CREATE\_CONTACTS\_TABLE = **"CREATE TABLE "** + ***TABLE\_CONTACTS*** + **"("** + ***KEY\_ID*** + **" INTEGER PRIMARY KEY,"** + ***KEY\_SPEED*** + **" TEXT,"** + ***KEY\_TIME*** + **" TEXT"** + **")"**;  
 db.execSQL(CREATE\_CONTACTS\_TABLE);  
}  
  
*// Upgrading database*@Override  
**public void** onUpgrade(SQLiteDatabase db, **int** oldVersion, **int** newVersion) {  
 *// Drop older table if existed* db.execSQL(**"DROP TABLE IF EXISTS "** + ***TABLE\_CONTACTS***);  
  
 *// Create tables again* onCreate(db);  
}

A Speed class was created that has all the setters and getters required. This class is communicating with the database by calling all the functions to get from the database. Within the Graph Activity, DBHelper db = **new** DBHelper(**this**); is written. This is calling the constructor of the DBHelper class, which is creating the database. A graphical user interface (GUI) of the database in table format can be viewed using DB Browser.

**Figure 11.7.1** 3D Speed database on DB Browser

# Website

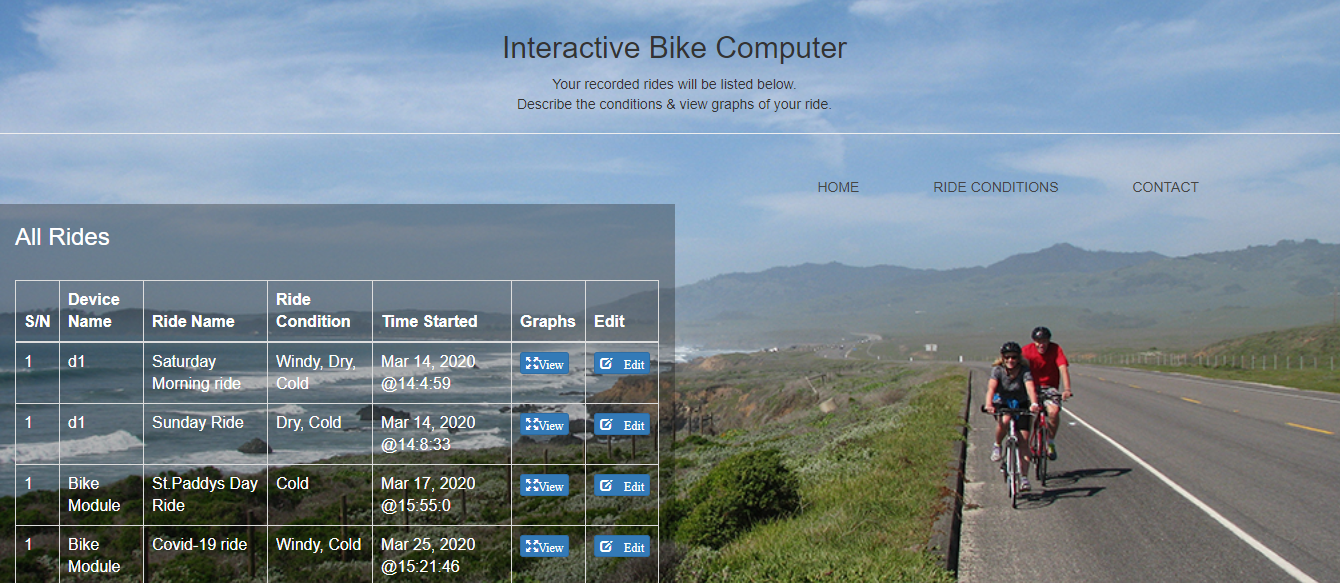
## Goal

The main goal for the website is to make the user want to view it. After completing a cycle, the user knows that the data they are interested in viewing is waiting there for them. Graphs are a lot easier to understand so there is a lot of focus on making sure that the data is shown in a very user-friendly manner. A custom domain name for the website helps to make it more realistic which can be found at interactive-bike.com.

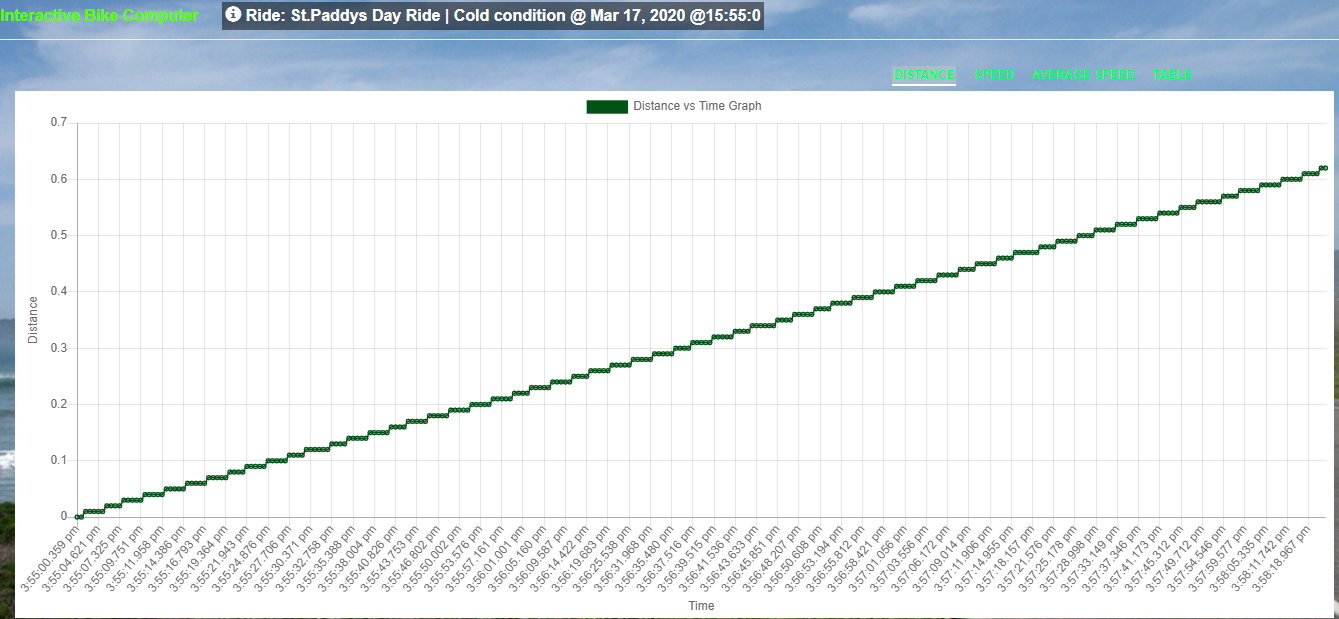
## Navigating the website

The user is welcomed to the home page with a list of all the rides they have completed to date. The user then has the option to edit their ride. This consists of naming the ride and adding weather conditions. Once ride conditions have been added, the user can view these conditions in a bar chart by clicking on ‘Ride Conditions’ (See Figure 11-4 below). Clicking on the ‘View’ button, which is located under ‘Graphs’ on the table, will open all the graphs specific to that ride.

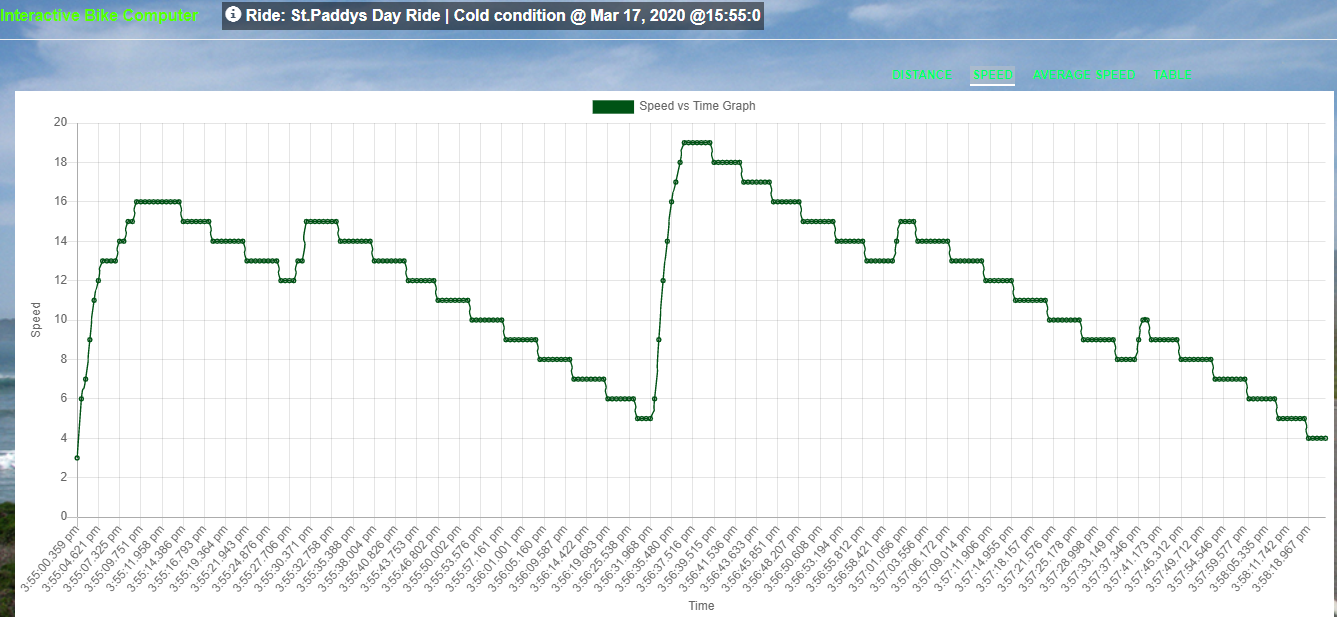
## Images of the website



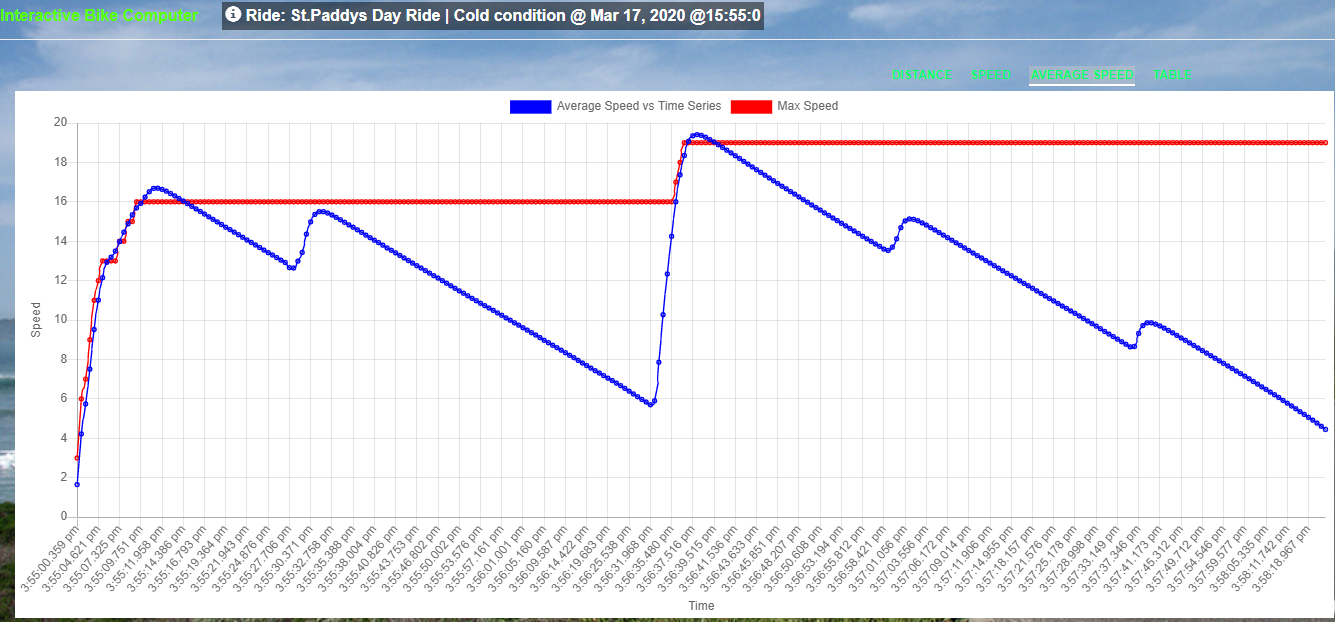
**Figure 12-3-1** Homepage of website



**Figure 12-3-2** Graph.js (Distance vs Time)



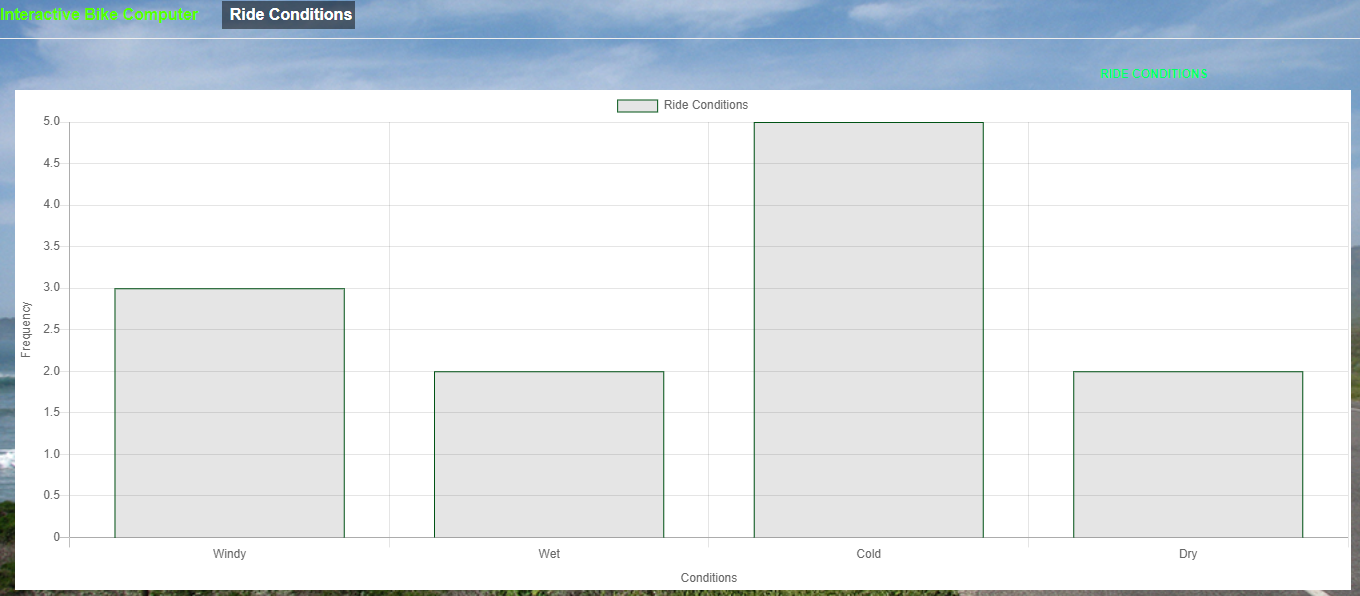
**Figure 12-3-3** Graph.js (Speed vs Time)



**Figure 12-3-4** Graph.js (Average Speed vs Max Speed)



**Figure 12-3-5** Graph.js (Average Speed vs Max Speed)



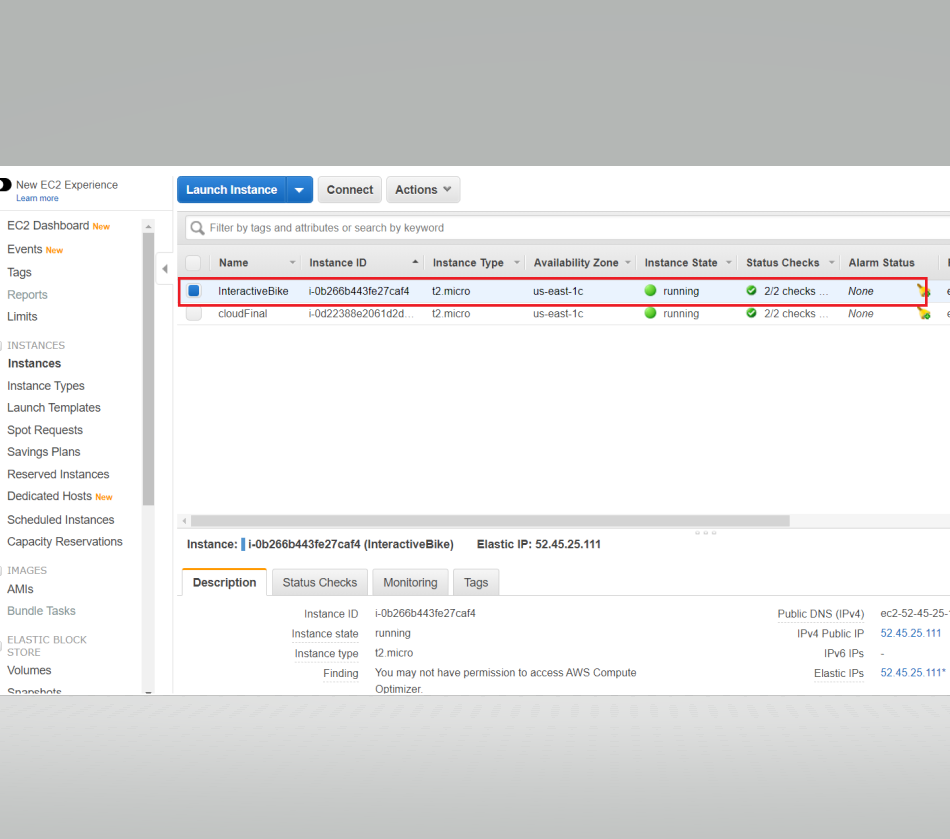
**Figure 12-3-6** Graph.js (Average Speed vs Max Speed)

## Amazon Web Services

Amazon Web Services (AWS) has many powerful features on its cloud computing platform. One of which is the EC2 (Amazon Elastic Compute Cloud) instance, which is a virtual server that is allowing me to run my website in a computing environment. AWS was chosen to deploy the website due to a working familiarity with the layout of the EC2 dashboard from the module completed this year, Cloud Computing. The guidance from the module and the abundance of documentation on the AWS site itself was extremely helpful. After creating the instance, inbound security rules had to be added, essentially opening ports for a TCP/IP connection.

### Connecting my website to AWS server

The server connects to the application via SSH, which stands for ‘Secure Shell.’ It is a protocol used to connect to a remote server/system securely. Amazon EC2 uses public-key cryptography to encrypt and decrypt login information. Public-key cryptography uses a public key to encrypt a piece of data, and then the recipient uses the private key to decrypt the data. The public and private keys are known as a key pair **[10]**. Public key cryptography enables you to securely access your instances using a private key instead of a password. This private key or key pair is saved as a .pem file, which is essentially a unique pathway from the server. PuTTY is used to establish a SSH connection. Using PuTTYgen, the unique .pem file is converted to a .ppk file that is required to secure the connection **[11]**. This sets up the connection on a Linux operating system. As the primary laptop is running Windows, WinSCP was used, which allows the secure copy of files over to a remote server.



**Figure 12-4-1** EC2 Dashboard

### Domain name (DNS)

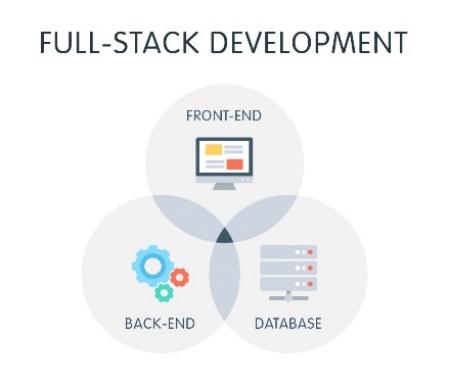
A domain name was purchased on GoDaddy to make the site feel a bit more realistic, as opposed to typing in an IP or EC2 address. It was simple to route the elastic IP address from AWS to the new domain name on GoDaddy. The website can be found at interactive-bike.com **[12].**

### Connecting NodeJS & MongoDB to AWS server

I an using a SSH connection with PuTTY. The command *‘$ sudo apt-get install nodejs npm mongodb’* installed both the runtime environment and database. To run the back-end I used the command *‘$ sudo nohup nodemon index.js </dev/null &’.* Nodemon constantly monitors the source code for any changes and restarts the server. The ‘&’ parameter starts keeps the backend running constantly.

## Full stack development

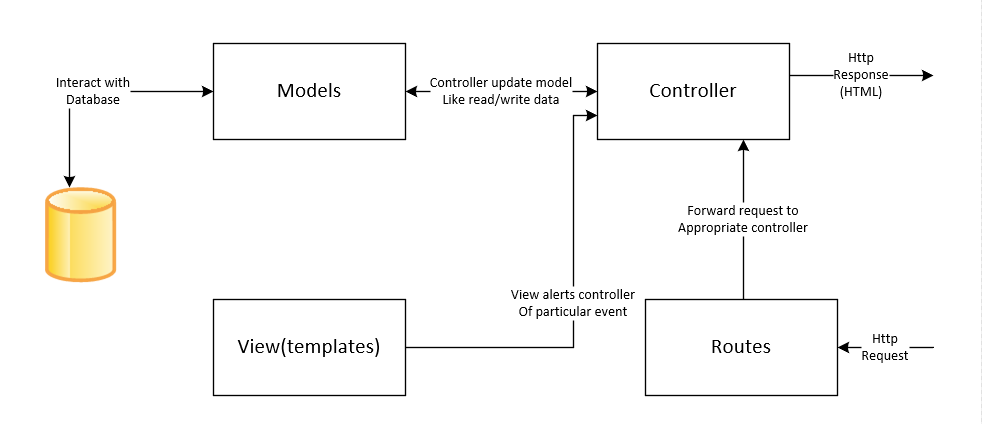
Full-stack development is the development of both frontend, backend, and databases within the same project. This project fits into this category. It was enjoyable and rewarding to be able to get a user interface running from building a database and backend system to power this frontend.



**Figure 12-5** Full stack development

### MVC Framework

Model-View-Controller (MVC) is the architectural pattern, which essentially divides the application into three main logical components, the model, the view, and the controller. The model receives all the data from the database and organizes it. The model does not depend on the controller or view. The view displays the model data and sends the user actions to the controller. The controller provides the model data to the view and interprets user actions, such as button clicks. The controller depends on the view and model. An extra element to the MVC framework is routes. Routes are essentially URL patterns associated with different pages. When a user enters a URL, the backend of the application tries to match that URL to one of the predefined routes **[13].**



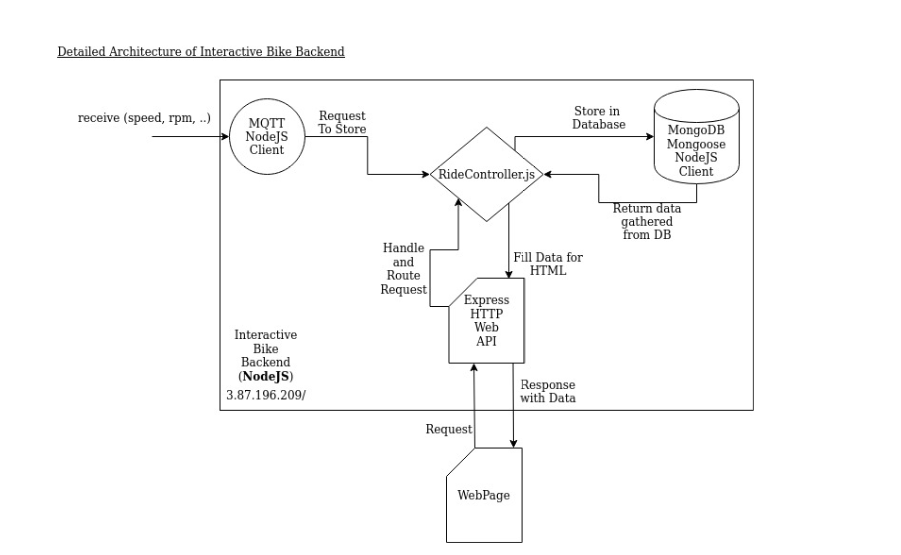
**Figure 12-5** MVC block diagram

#### Advantages of MVC

MVC helps avoid complexity by dividing the application into four main units. Those being model, view, controller, and routes. It was also a desire to implement asynchronous techniques, which MVC supports.

## Backend

The backend of the project is a Node.js application working alongside an Express.js framework and a MongoDB database. Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine and is used for easily building fast and scalable network applications. Express.js is a modular web framework for Node.js. It essentially simplifies the development process of server-side applications.



EJS

Templating engine

Render the data

### Database & Models

The database used is MongoDB, with Mongoose used to access the library data. Mongoose acts as a front end to MongoDB, which is a NoSQL database that uses a documented-oriented data model (ODM). A “collection” of “documents” in a MongoDB database is analogous to a “table” of “rows” in a relational database. This combination of ODM and database is very popular in the Node.js community, mainly because the document storage system is very similar to JSON, thus making things a lot easier for the JavaScript side of things **[14]**.

After importing the mongoose module

var mongoose = require('mongoose');

A rideModel.js was created, which is defined using a Schema interface. The Schema allows the definition of the fields to store in each document in the database along with validated requirements and default values.

var rideSchema = mongoose.Schema({

    d\_id: String,

    s\_time: Number,

    name: String,

    cond: String,

    data:   [{

            speed: {

                type: String,

                required: true

            },

            rpm: {

                type: String,

                required: true

            },

            time: {

                type: Date,

                default: Date.now

            },

            distance: String,

            average: String,

            maximum: String,

    }]

});

All required data is passed through the Schema which is then compiled into models using the mongoose.model method(). It is then exported to create the model.

var Ride = module.exports = mongoose.model('ride', rideSchema);

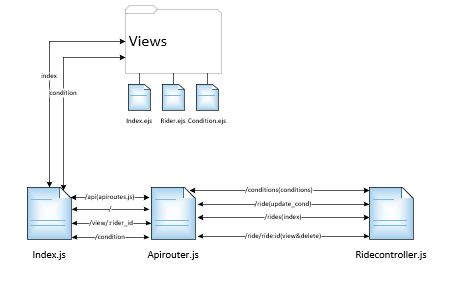
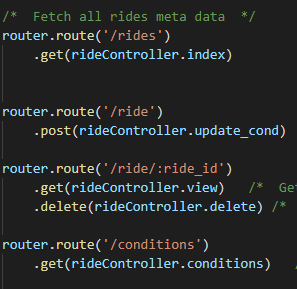
The model can then be ‘required’ immediately in the rideController.js file.

### Routes

A file, api-routes.js, forwards the requests to the rideController.js function. A route is a section of Express code that associates an HTTP verb (GET, POST, PUT, DELETE, etc.), a URL path/pattern, and a function that is called to handle that pattern. The express.Router() middleware allows grouping the route handlers for a specific part of the website and accessing it via a common route-prefix.

let router = require('express').Router();

REST APIs, get(), post(), and delete() requests are used, which will configure the Router object to be exported to the rideController.js file, depending on the request.

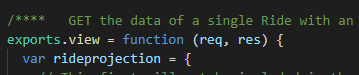


**Figure 12-6-1** API-routes file connection

### Controllers

In terms of controllers, a file called rideController.js handles the user interaction—interpreting the mouse and keyboard inputs from the user, which in turn informs the model and view to change accordingly. The controller file gets the database Schema from rideModel.js that interacts with the MongoDB database, for read and write data.

This is how the exports controller logs into api-routes.js

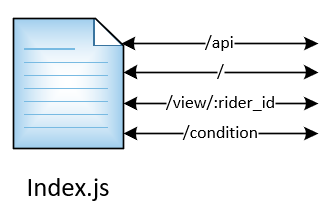


### View

Views is the directory where the template files are located. A templating engine enables use of static files in the application. At runtime, the template engine replaces variables in a template file with actual variables and transforms the template into an HTML file sent to the client. The templating engine is EJS. EJS stands for Embedded JavaScript and is used with Express.js. It lets the application directly render HTML and JSON dynamically using Express.js. EJS was chosen as there was no need for a frontend templating framework because the application is quite simple.

Converting the engine to EJS

app.set('view engine', 'ejs');



We can see from the above code snippet and diagram of the index.js file, which route renders in the view folder and which route is selected.

### MQTT website connection

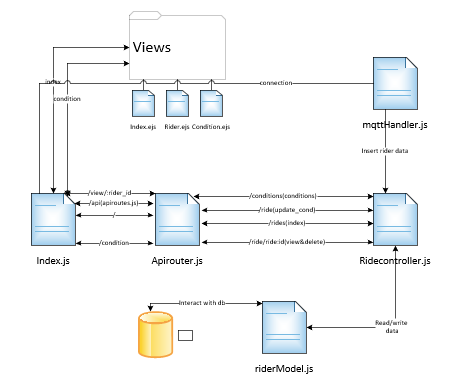
We configure MQTT library to insert the rider data into rideController.js. All the broker credentials are stated in the mqttHandler.js file. The data is then parsed into JSON. This file is then imported into index.js and it checked for a valid connection or not.

if (!mqttHandler.connected()) {

     console.log("Still not connected to MQTT broker...");

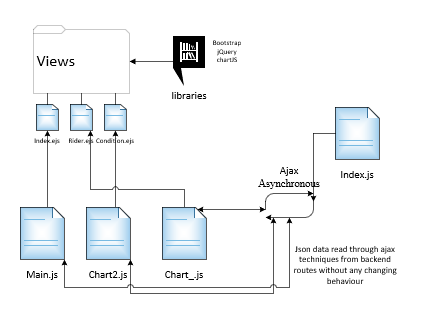
}

### Backend diagram



## Frontend

The main goal of the frontend was to make it very easy to use for the cyclist. To be able to give them the confidence that they can access the website and be able to navigate through it smoothly. One way this became possible was by using jQuery in parallel with AJAX.



### jQuery & Ajax

jQuery is a JavaScript library designed to simplify HTML document traversal and manipulation, and event handling along with Ajax. This easy-to-use API works across multiple browsers. In terms of this project, it made working with JavaScript a lot easier. It wraps common tasks into methods that can be called in a single line of code. Ajax works with jQuery on the client-side to create asynchronous web development. Essentially, it allows sending and receiving data from the server asynchronously without interfering with the display and behavior of the existing page. jQuery() can also be written as $.

$(document).ready(function(){

This specifies the call back function which will be executed when the page is loaded. Below we can see ajax asynchronously getting JSON data from the specific route, ride\_id.

$.ajax({

    type: 'GET',

    url: '/api/ride/' + ride\_id,

    dataType: 'json',

    error: function (data) {

        console.log("error");

    },

### Chart.js

Line and bar charts are displayed using a library called Chart.js. This library is very easy to use when implementing the interactive graphs **[15]**.

The chart is set to the desired type, then data is added for what the chart should display. The graph is responsive so that it will fit data from the whole ride no matter the size. Using a scales function, dividing the x-axis and y-axis displays the desired formats.

type: 'line',

    data: data\_2,

    options: {

        fill: false,

        responsive: true,

        scales: {

            xAxes: [{

                type: 'time',

                time: {

                            displayFormats:{

                                'hour' : 'hh:mm;ss',

                                }

                        },

                distribution: 'series',

                display: true,

                scaleLabel: {

                    display: true,

                    labelString: "Time",

                }

            }],

A library called moment.js allows parsing and display of dates and times in JavaScript **[16]**. Chart.bundle.min.js allows addition of a colour library and customization of the line to make the graphs more aesthetically pleasing.

datasets: [{

        fill: false,

        label: 'Speed vs Time Graph',

        data: speed\_series,

        borderColor: '#005315',

        borderWidth: 1.4,

        lineTension: 1,

        }]

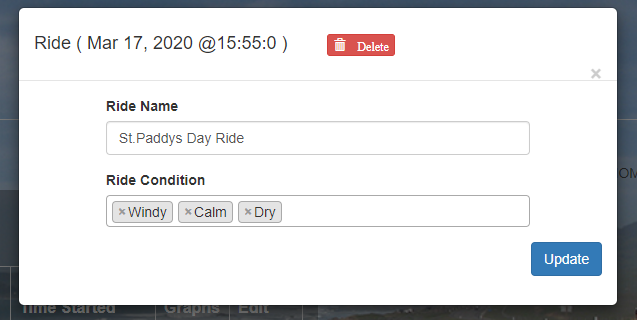
### Bootstrap

Bootstrap is a powerful CSS framework for developing responsive and mobile-first websites. Bootstrap requires jQuery to function. Bootstrap helps create responsive web applications at a fast rate. Responsive means the features on the website will change size depending on if the window is fully open or not. Bootstrap also offers countless glyphicon components **[17].**

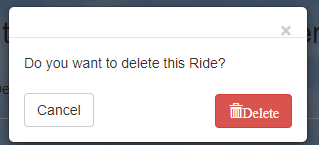
**Figure 12-7-3-1** Glyphicon components

The application also uses a bootstrap modal, which is essentially a pop-up window. Only content on the modal window can be configured unless you exit the window. A modal makes the experience for the user a lot better than loading another page, along with it being a lot faster.



**Figure 12-7-3-2** Modal pop-up window

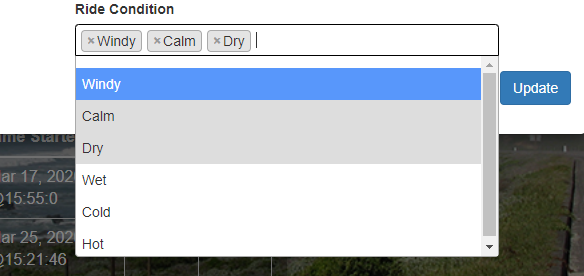
A confirmation modal was added, which asks the user if they are sure they want to delete all data for that specific ride.



**Figure 12-7-3-4** Confirmation modal pop-up window

### Select2.js

The Select2.js library is another jQuery-based replacement for select boxes. It is also very user-friendly. Using this library gives the user the quickest chance to describe their ride conditions **[18]**.



**Figure 12-7-4-1** Confirmation modal pop-up window

As you can see from *Figure 12-7-4-1* above, the user is given the choice of different ride conditions. This is ideal for the user to remember details of their cycle that they can look back on sometime in the future.

# System Integration

The system integration aspect of the project has been successful. There were no real problems with connecting the Android application to the MQTT Broker. It also gave the ability to display real-time data on a mobile device. The next step in system integration was storage and display of data on the site.

One obstacle during site creation happened during the configuration of the rides by date stamps. If a cyclist clocked two rides on the same day, the site would combine them into one ride in error. After going back to the drawing board and seeking opinions from lecturers and classmates, each ride was configured with a unique ID and given a 20 second time threshold. If no new data is published to the app after 20 seconds, the ride is marked as complete. Any data following will be considered part of a new ride.

Apart from this, there were no major roadblocks. This can be attributed to the sheer amount of documentation and access to readymade libraries that are available to get anyone started with full-stack development.

# Conclusion

The main goal of this final year project was to design and build a fully functional bike computer. This bike computer would allow a cyclist to view real-time data on their mobile device and also provide a website where data can be easily accessed at any time.

Research began with the selection of the Hall effect sensor over a reed switch as there are no moving parts, and it can operate at high speed required when working with a bike. The module, Mobile App Development, guided research for implementation of the Android application. The module helped with understanding the basics of mobile development and got this project successfully off the ground. Site research was aided by not only the Cloud Computing module but the numerous online tutorials that helped make it easy to understand how to implement full-stack development.

Many new skills have been acquired during the process ranging from how to embed JavaScript with EJS, simplifying JavaScript with JQuery, adding designs using Bootstrap, deploying applications with AWS, and displaying data asynchronously with Ajax.

Overall, this project was a success and one that was of personal interest and a great learning experience.

# References

[1] **Cyclometer - History of bike computers.** Wikipedia**.** Available: <https://en.wikipedia.org/wiki/Cyclocomputer>

[2] **Raspberry Pi.** Available: <https://www.raspberrypi.org/products/raspberry-pi-4-model-b/>

[3] **Cron task.** Raspberry Pi. Available: <https://www.raspberrypi.org/documentation/linux/usage/cron.md>

[4] **Hall effect sensor.** Electronics Tutorials. Available: <https://www.electronics-tutorials.ws/electromagnetism/hall-effect.html>

[5] **CloudMQTT.com.** Cloud based MQTT broker. Available: <https://www.cloudmqtt.com/docs/index.html>

[6] **Google Play Console Platform.** Android Studio developer’s documentation. Available: <https://developer.android.com/distribute/console>

[7] **MapView.** Google APIs for Android. Available: <https://developers.google.com/android/reference/com/google/android/gms/maps/MapView>

[8] **Android Widgets.** GitHub for Analog Speedometer. Available:  <https://github.com/ntoskrnl/AndroidWidgets>

[9] **Android SQLite Tutorial.** Developed by JavaTpoint**.** Available:  <https://www.javatpoint.com/android-sqlite-tutorial>

[10] **Amazon EC2 Key Pairs.** Amazon Web Services. Available: <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-key-pairs.html>.

[11] **PuTTYgen.** Key Generator for PuTTY on Windows for SSH connection. Available: <https://www.ssh.com/ssh/putty/windows/puttygen>

[12] **Connecting an EC2 instance with a GoDaddy Domain**. Medium**.** Available: <https://medium.com/progress-on-ios-development/connecting-an-ec2-instance-with-a-godaddy-domain-e74ff190c233>

[13] **MVC Application - By James Kolce, Nilson Jacques, March 23, 2020.** Available: <https://www.sitepoint.com/node-js-mvc-application/>

[14] **Using a Database (with Mongoose) – by MDN contributors, Mar 21, 2020**. Available: <https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose>

[15] **Chart.js.** MIT license. Available: <https://www.chartjs.org/>

[16] **Moment.js.** MIT license. Available: <https://momentjs.com/>

[17] **Glyphicon components.** Bootstrap. Available: <https://getbootstrap.com/docs/3.3/components/>

[18] **Select2.js.** The jQuery replacement for select boxes. Available: <https://select2.org/>

[19] **Stackoverflow - Daniel Nugent, July 9th 2017.** How to get current Location in GoogleMap using FusedLocationProviderClient.Available: <https://stackoverflow.com/questions/44992014/how-to-get-current-location-in-googlemap-using-fusedlocationproviderclient>

# Appendices

## Bill of Materials

1x Hall Effect Sensor (Keyes module)

2x Raspberry Pi 4

1x Magnet

## GitHub links

<https://github.com/SHeavey/FYPBikeComputer.git>

<https://github.com/SHeavey/BikeAppFinal.git>